//Semih Balki - 19010

a)

Identify the subproblems:

-The minimum number of traffic lane change of a subset of objects {1, 2, ..., i}, i number of row

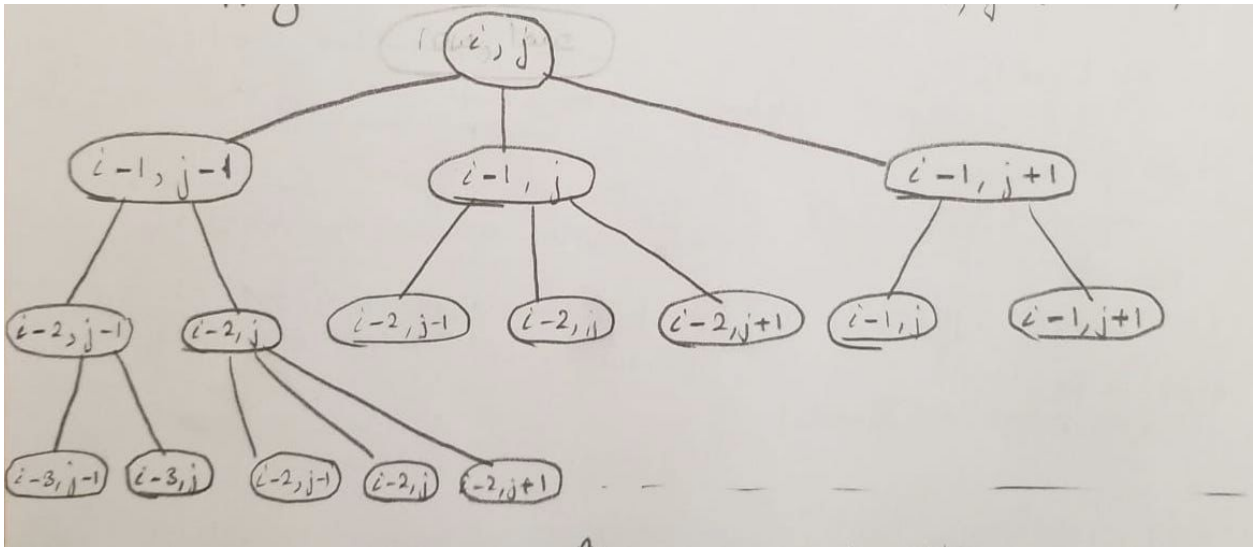
Optimal substructure property:

-Let i be an element of {1, 2, …, row}, j be an element {1, 2, …, lane}

And let k be the minimum number of traffic lane change for i and j, i < row and j < lane.

Then any value smaller than k is also solution for i and j.

Overlapping computations:



Recursive formulation:

Let A be a matrix(2D array) that contains the information of the obstacles at the traffic lane problem.

$$A[i, j] = \begin{cases} 0, & i = A.rowlength \\ 1, & A[i, j] = 1 \\ \min\{func(i + 1, j - 1) + 1, func(i + 1, j), func(i + 1, j + 1) + 1\}, & otherwise \end{cases}$$

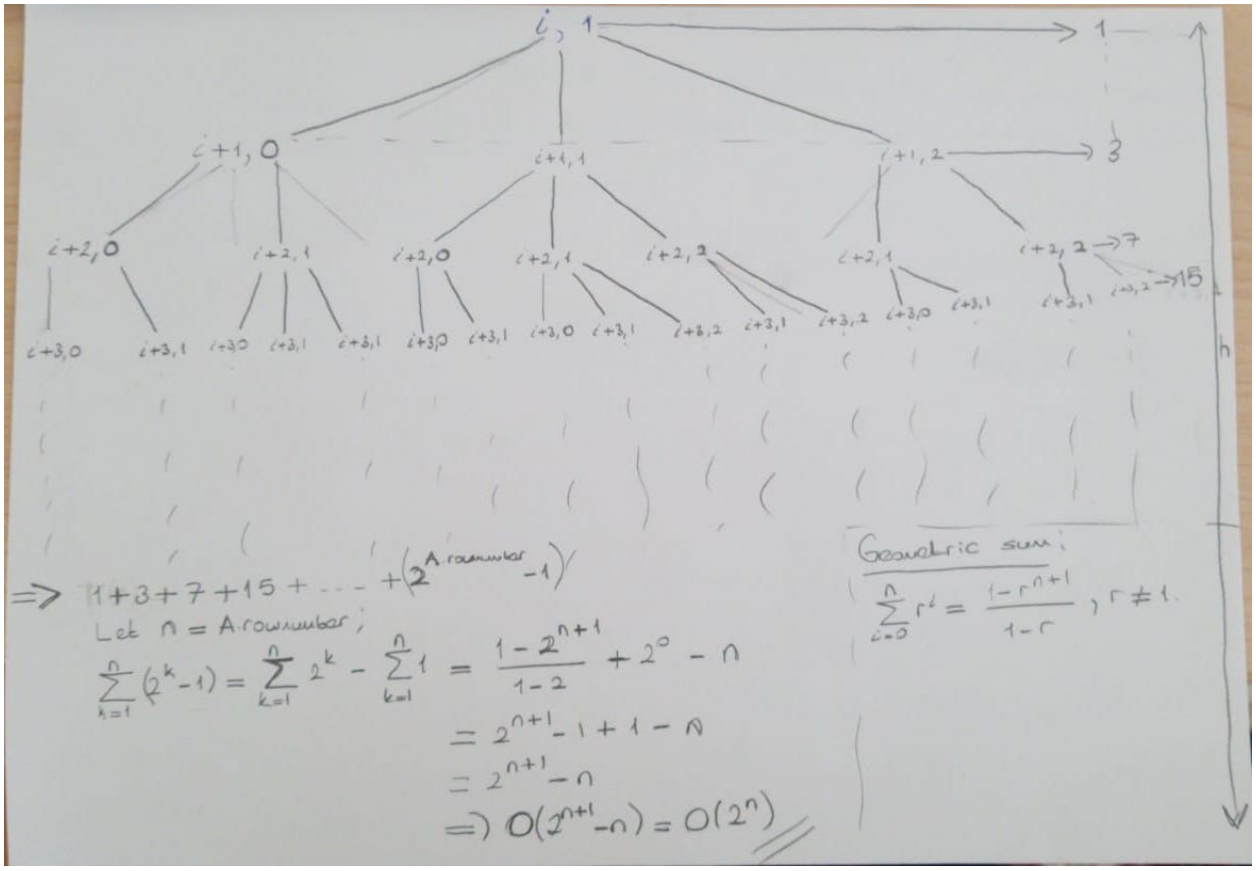
b)

```
NAIVE(row, lane)
  if row == n
        return 0
  else if A[row, lane] == 1
      return 1
  else
      if lane - 1 > 0 and lane + 1 < len(A[0])
         return min(NAIVE(row + 1, lane - 1) + 1, NAIVE(row + 1, lane), NAIVE(row + 1, lane
+ 1) + 1)
      else if lane == 0
         return min(NAIVE(row + 1, lane), NAIVE(row + 1, lane + 1) + 1)
      else
         return min(NAIVE(row + 1, lane - 1) + 1, NAIVE(row + 1, lane))
```

Complexity Analysis:

The worst case happens when the lane of the car is at 1:

Let A.rownumber > i = car.row >= 0.

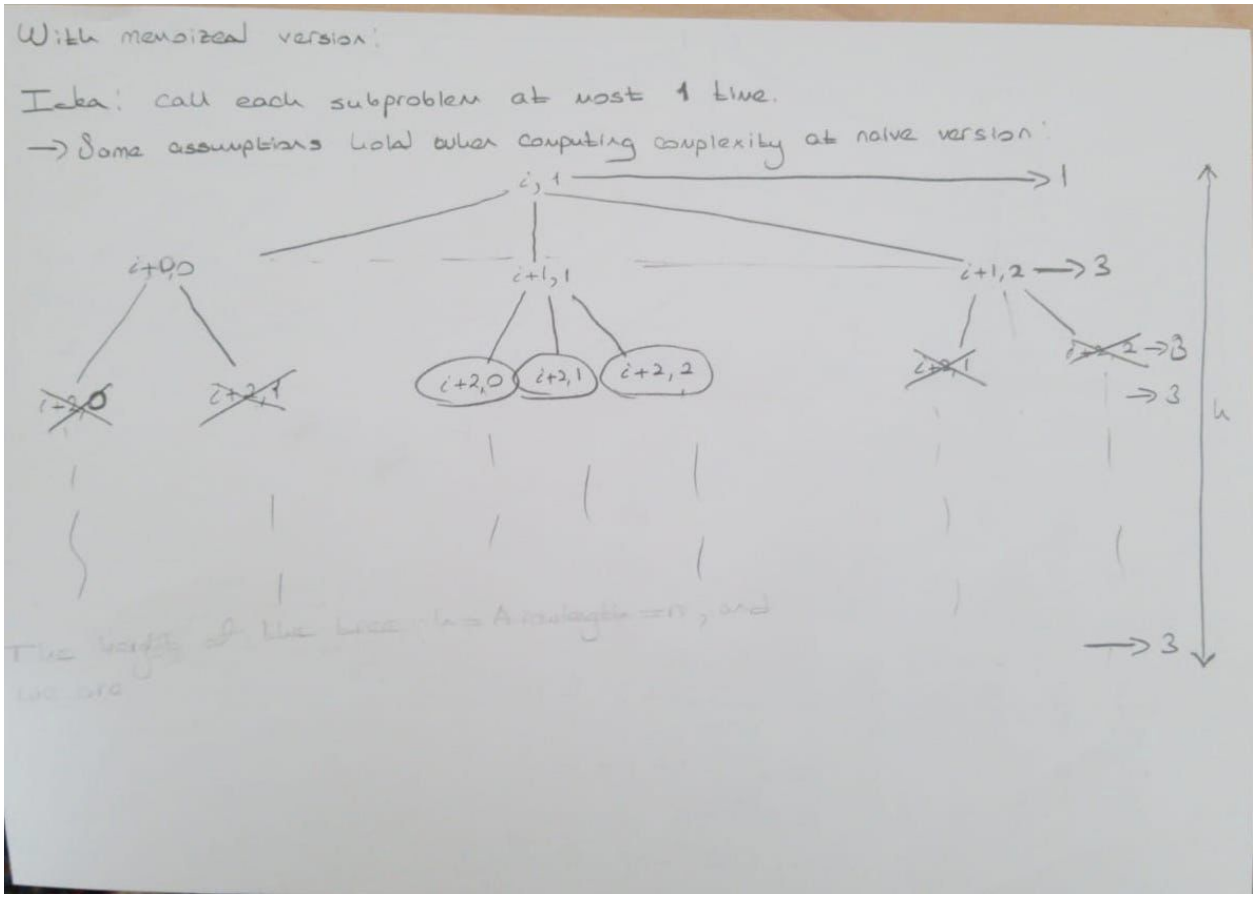


Space Complexity: none, since we just created 2d array for roadway matrix which is not specific for this function.

c)

```
Top_down_with_Memo(row, lane)
   if row == n
      return 0
   else if memo[row, lane] != -1
      return memo[row, lane]
   else if A[row, lane] == 1
```

```
        return 1
    else:
        if lane - 1 > 0 and lane + 1 < len(A[0])
            memo[row, lane] = min(Top_down_with_Memo(row + 1, lane - 1) + 1,
Top_down_with_Memo(row + 1, lane), Top_down_with_Memo(row + 1, lane + 1) + 1)
        else if lane - 1 < 0
            memo[row, lane] = min(Top_down_with_Memo(row + 1, lane),
Top_down_with_Memo(row + 1, lane + 1) + 1)
        else
            memo[row, lane] = min(Top_down_with_Memo(row + 1, lane - 1) + 1,
Top_down_with_Memo(row + 1, lane))
    return memo[row, lane]
```



The height of the tree is A.rowlength = n, and at each level there is 3 problems. Therefore, O(3.n) = O(n)

Space complexity is O(3.n) = O(n) since we created a matrix for memoization.

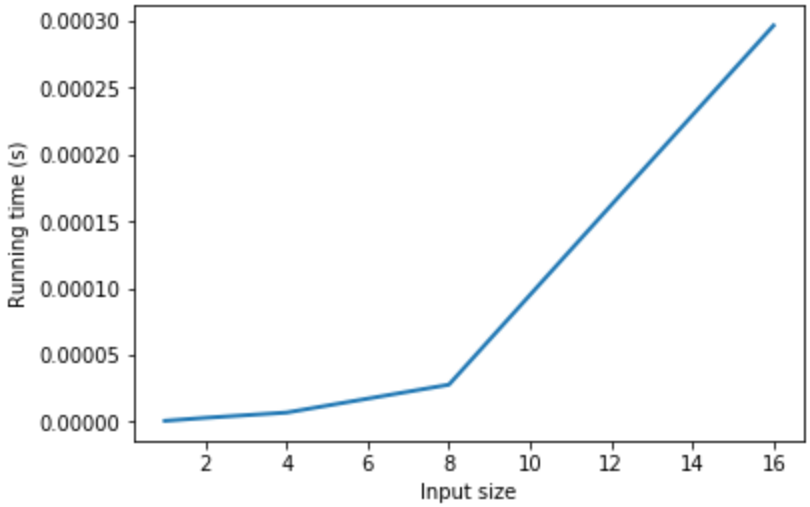
d)

```
Bottom_up_iterative(row, lane)
   if row == n
      return 0
   elif memo[row, lane] != -1
      return memo[row, lane]
   elif A[row, lane] == 1
      return 1
   else
      for i in range(n)
         if lane - 1 > 0 and lane + 1 < len(A[0])
            memo[i, lane] = min(Bottom_up_iterative(i + 1, lane - 1) + 1, Bottom_up_iterative(i +
1, lane), Bottom_up_iterative(i + 1, lane + 1) + 1)
         elif lane - 1 < 0
            memo[i, lane] = min(Bottom_up_iterative(i + 1, lane), Bottom_up_iterative(i + 1, lane
+ 1) + 1)
         else
            memo[i, lane] = min(Bottom_up_iterative(i + 1, lane - 1) + 1, Bottom_up_iterative(i +
1, lane))
   return memo[i, lane]
```

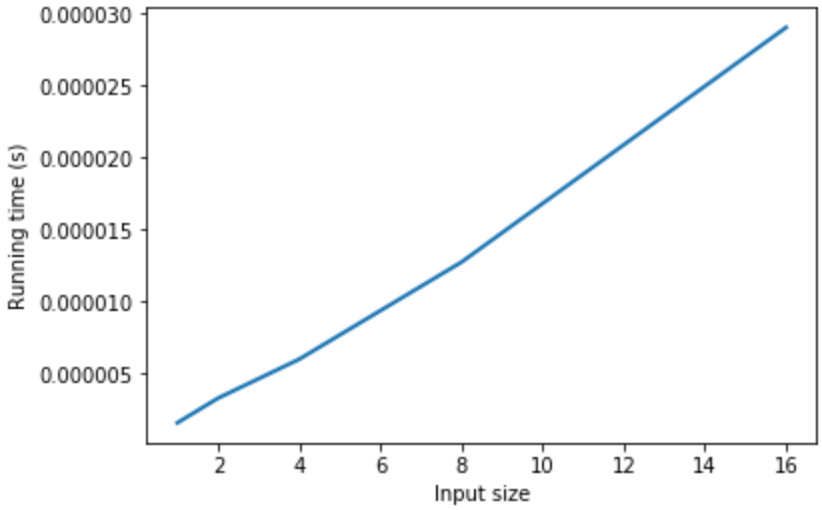
Time Complexity: O(n), same as top down recursive algorithm.
Space Complexity: O(n), same as top down recursive algorithm.

e)

The following graph is the result of NAIVE recursive algorithm which is not surprising since we expected the complexity as exponential so the graph does.



The following graph is the result of Top down with memoization algorithm which is not surprising since we expected the complexity as linear so the graph does.



Note that: You can view the code from here:
https://nbviewer.jupyter.org/gist/ftalay/c32af1c019b14f9b2ba051ee3974089c
(I will also submit as .py file of the code, shared the link to visualize better)